

Customizing Static Analysis for C and C++

Coverity Extend is used for:

- Finding custom or domain specific defects unique to your code
- Complying with company or industry standards such as MISRA
- Automating parts of the manual code review

Sample Checkers Coverity Customers Have Built:

- Disable interrupts locally rather than globally
- Enforce custom security policies
- Use memory mapped I/O rather than copying
- Enforcing naming conventions

Coverity Extend

Coverity Extend is an easy-to-use **Software Development Kit (SDK)** that **allows developers to detect unique defect types in C and C++ code.**

Coverity Extend comes with all the out-of-the-box checkers in Coverity Prevent and leverages the Coverity Analysis Engine. To use Coverity Extend, developers utilize the simple but powerful SDK to write checkers in C++. Next, Coverity Extend analyzes the code to pinpoint the developer-defined defects. All the discovered defects are consolidated in the Coverity Prevent™ code browser alongside standard defects. All of the extensions built by developers are stored in a **Custom Checker Library** so developers can build, execute and store an unlimited number of checkers.

Using Coverity Extend, Coverity's customers have built custom checks to find and repair defects relating to concurrency, exception handling, and more.

Benefits

- Improve software quality by finding defects that are most important to your development organization
- Extend's flexibility and ease of use allow you to better model the rules you need to enforce
- Reduce compliance costs by adhering to corporate and industry coding standards
- Reduce the labor overhead associated with manual code inspections
- Automate the code inspection process to address custom development conventions

Enforcing Industry Best Standards

Coverity Extend can also help developers achieve compliance with corporate or industry standards. Coverity Extend allows you to add targeted stylistic rules that check only those conventions that are important to your organization without adding another different, separate product to the development process that will enforce unrealistic stylistic constraints.

Example: simple check to identify assignment operations within "if" conditions

```
300 #include "extend-lang.h"
301 START_EXTEND_CHECKER( no_assign, simple );
302 ANALYZE_TREE()
303 {
304     if (MATCH (Or( If( ( _ = _ ) != 0 ), If( ( _ = _ ) == 0))) ) {
305         OUTPUT_ERROR("within if.");
306     }
307 }
308 END_EXTEND_CHECKER();
309 MAKE_MAIN ( no_assign );
```

How Does it Work?

Writing Coverity Extend checks only requires C++ programming skills. Once developers compose an extension, it hooks into the Coverity Analysis Engine which executes the checks across the entire code base.

Building an Extend check involves a three-step process:

1. **Define a rule** — The developer begins the process of creating a check by defining a sequence of source code actions along a code path that could trigger a defect. A memory leak, for example, involves a memory allocation event, no freeing event, and finally a return from a function. To simplify the process for the developer, Coverity

Extend provides a very simple library of macros and templates to perform highly complex analysis functions via the Coverity analysis engine.

2. **Specify pattern matches** — The developer then specifies a pattern to match against the line and artifact in the source code. At the simplest level, matches can be textually based, such as a function name. However, very complex defects can be captured since nearly any type of pattern can be expressed as an Extend rule.

3. **GUI integration** — Coverity Extend provides a standard set of customizable routines for creating error messages and GUI integration.

Path analysis example: detecting performance degradation caused by a blocking call

```
200 #include "extend-lang.h"
201 enum fun_state_t {
202     UNLOCKED = 0,
203     LOCKED = 1
204 };
205 START_EXTEND_CHECKER( block_check, int_store );
206 ANALYZE_TREE()
207 {
208     Fun locking_fun("lock");
209     Fun unlocking_fun("unlock");
210     Fun blocking_fun("fopen");
211     if ( MATCH(locking_fun) ) {
212         SET_STATE(LOCKED);
213     } else if ( MATCH(unlocking_fun) ) {
214         SET_STATE(UNLOCKED);
215     } else if ( MATCH(blocking_fun) ) {
216         if ( GET_STATE() == LOCKED ) {
217             COMMIT_ERROR("Called fopen within a locking context.");
218         }
219     }
220 }
221 END_EXTEND_CHECKER();
222 MAKE_MAIN( block_check )
```

Supported Platforms

- Linux
- FreeBSD
- Windows
- Solaris Sparc
- Solaris X86

Supported Compilers

- Sun CC
- MS Visual Studio
- gcc
- g++
- Arm compilers
- Intel compiler for C/C++
- Intel MSA compiler
- Wind River Diab compiler
- TI Code Composer C compiler
- Green Hills compiler
- IAR compiler
- PICC compiler
- Support for other ANSI C compatible compilers available upon request

Deployment Options

Developers can leverage Coverity Extend in two ways:

1. Do-it-yourself—Developers can opt to write checkers themselves.
2. Coverity Professional Services—Coverity's engineers can create custom checkers for you.

About Coverity

Since 2002, Coverity Inc. has provided source code analysis solutions that are a leap forward for software engineers who build reliable, secure systems. Based on years of research at the Computer Systems Laboratory at Stanford University, Coverity Prevent is now providing customers such as Juniper Networks, Wind River, NASA, Palm, France Telecom and nVidia Corporation a revolutionary way to build more reliable code while simultaneously shortening the time to market.

Coverity in Action:
<http://linuxbugs.coverity.com>
<http://www.coverity.com/>
sales@coverity.com

Coverity Inc. Headquarters
185 Berry St. Suite 2400
San Francisco, CA 94107 USA
Phone: (800) 873-8193



© 2006 Coverity, Inc.